

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Fall Semester, 2001

**Quiz I**

**Closed Book – one sheet of notes**

Separately, we have distributed an answer sheet. You may use the space on the exam booklet for whatever temporary work you find useful, but you **MUST** enter your answers into the answer sheet. **Only this will be graded.** Each problem that requires an answer has been numbered. Place your answer at the corresponding number in the answer sheet.

Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice.

The answer sheet asks for your section number and tutor. For your reference, here is the table of section numbers.

Section	Time	Location	Rec. Instructor	Tutors
1	10:00	26-328	Randy Davis	Attila Kondacs
2	11:00	26-328	Randy Davis	Attila Kondacs/ Ben Vandiver
3	11:00	8-302	Tony Eng	Jeff Klann
4	12:00	8-306	Tony Eng	Jeff Klann/ Ben Vandiver
5	1:00	26-204	Konrad Tollmar	James Robertson
6	2:00	26-204	Konrad Tollmar	James Robertson/Connie Cheng
7	11:00	36-156	Paul Penfield	Jesse Pavel
8	10:00	36-155	Duane Boning	Connie Cheng
9	12:00	36-153	Paul Penfield	Jesse Pavel

**IMPORTANT: Write your name on EVERY page of the answer sheet!**

**Part 1: (16 points)**

For each of the following expressions or sequences of expressions, state what value is returned as the result of evaluating the final expression in each set. Each part is independent—in other words, for each part, state what value would be returned if that expression or sequence of expressions were the only expressions evaluated in a newly initialized Scheme system.

If the evaluation results in an error, please state in general terms what kind of error (e.g. you might write “error: wrong type of argument to procedure”). If the evaluation returns a built-in procedure, write `primitive procedure`. If the evaluation returns a user-created procedure, write `compound procedure`.

In addition, for all non-error values, also write the type of the returned value, using the notation introduced in lecture.

**Question 1.**

```
(+ 1 2)
```

**Question 2.**

```
(1 + 2)
```

**Question 3.**

```
+
```

**Question 4.**

```
((lambda (* a b) (b * a)) 2 3 -)
```

**Question 5.**

```
(lambda (a b)
  (if (> a b) a b))
```

**Question 6.**

```
(lambda (x) (lambda (y) (expt y x)))
```

**Question 7.**

```
((lambda (a b)
  (if (> a b) + *))
  5
  2)
3
1)
```

**Question 8.**

```
(define cube (lambda (x) (* x x x)))
(define (compose f g)
  (lambda (x) (f (g x))))
(list ((compose cube (lambda (x) (+ x 1))) 2)
      ((compose (lambda (x) (+ x 1)) cube) 2))
```

**Part 2: (24 points)**

A useful capability is to sort a collection of elements, that is to arrange them in order based on some criteria. A simple version of this is sorting a **list of non-negative numbers** in increasing order. One way to do this is to find the smallest element in the list, keep it, and remove that element from the list, then repeat the process on the new list. Below is a procedure for accomplishing this:

```
(define (sort lst)
  (if (null? lst)
      nil
      (let ((best (once-through lst))
            (rest (remove best lst)))
        (cons best (sort rest))))))
```

The idea behind `once-through` is that it should pass over the list once, and return the smallest element. The idea behind `remove` is that it should return a new version of its second argument, with the first instance of the first argument absent from it (note that there might be multiple instances of an element). For example:

```
(remove 3 (list 1 2 3 4 3 4 5))
;Value: (1 2 4 3 4 5)
```

```
(remove 3 (list 1 2 3 4))
;Value: (1 2 4)
```

```
(remove 3 (list 1 2 4))
;Value: (1 2 4)
```

Here is a partial implementation of `once-through`.

```
(define (once-through lst)
  (if (null? lst)
      nil
      (aux (car lst) (cdr lst))))
(define (aux best rest)
  EXP)
```

**Question 9:** The procedure `aux` needs to be completed by providing code for `EXP` (write this without using the Scheme procedure `max` or `min`).

**Question 10:** Write the procedure `remove`, just using simple predicates and list operations like `car`, `cdr`, `cons`.

**Question 11:** What is the order of growth in time (as measured by the number of steps in the computation) and in space (as measured by the maximum number of deferred operations) for `once-through`, given a list of length `n` as input? Please select from

- A:  $O(1)$
- B:  $O(n)$
- C:  $O(2^n)$
- D:  $O(n^2)$
- E:  $O(\log n)$
- F: something else

**Question 12:** What is the order of growth in time and space for `remove`?

**Question 13:** What is the order of growth in time and space for `sort`?

**Part 3: (26 points)**

You want to write a procedure for reversing the elements of a list, for example, suppose `test` has the structure:

```
(1 (2 (3 4) 5) (6 7) 8)
```

then we want

```
(reverse (list 1 2 3 4))  
;Value: (4 3 2 1)
```

```
(reverse test)  
;Value: (8 (6 7) (2 (3 4) 5) 1)
```

**Question 14.** Write the procedure `reverse` so that it gives rise to a recursive process. Your procedure should handle an empty list correctly, should handle the case of a single element (i.e. something that is not a list) by returning the same element (for use in subsequent parts), as well as the general case of a non-empty list.

**Question 15.** Write a procedure `ireverse` that accomplishes the same thing, but in an iterative process.

As you have noticed, `reverse` only reverses the top level structure of a list, so when given a tree structure such as `test`, it does not recursively reverse the elements of the list. `Deep-reverse` is intended to accomplish a complete reversal of the tree, for example

```
(deep-reverse test)
;Value: (8 (7 6) (5 (4 3) 2) 1)
```

Below are some possible outcomes of applying different implementations of `deep-reverse` to `test`:

A: (8 (7 6) (5 (4 3) 2) 1)

B: (8 (6 7) (2 (3 4) 5) 1)

C: (8 (7 6) (5 (3 4) 2) 1)

D: (8 7 6 5 4 3 2 1)

E: infinite loop

F: some other error

G: some other value

For each of the following possible implementations of `deep-reverse` indicate the outcome that matches.

**Question 16.**

```
(define (deep-reverse tree)
  (if (or (null? tree) (not (pair? tree)))
      tree
      (map deep-reverse (reverse tree))))
```

**Question 17.**

```
(define (deep-reverse tree)
  (if (or (null? tree) (not (pair? tree)))
      tree
      (map reverse (deep-reverse tree))))
```

**Question 18.**

```
(define (deep-reverse tree)
  (if (or (null? tree) (not (pair? tree)))
      tree
      (append (deep-reverse (cdr tree))
              (list (deep-reverse (car tree))))))
```

**Question 19.**

```
(define (deep-reverse tree)
  (if (or (null? tree) (not (pair? tree)))
      tree
      (append (reverse (cdr tree))
                (list (reverse (car tree))))))
```

**Question 20.**

```
(define (deep-reverse tree)
  (if (or (null? tree) (not (pair? tree)))
      tree
      (append (deep-reverse (cdr tree))
                (list (reverse (car tree))))))
```

**Part 4: (14 points)**

Consider the following procedures:

```
(define (addfactor f)
  (lambda (x) (* x (f x))))

(define (again f n)
  (if (= n 1)
      f
      (lambda (g) ((again f (- n 1)) (f g)))))

(define (id x) x)

(define (more x) (+ x 2))
```

**Question 21.** What value is returned by the following expression

```
((addfactor id) 7)
```

**Question 22.** What value is returned by the following expression

```
((addfactor more) 3)
```

**Question 23.** What value is returned by the following expression

```
((again addfactor 2) id) 3)
```

**Question 24.** What value is returned by the following expression

```
((again addfactor 2) more) 5)
```

**Part 5: (20 points)**

In a previous question, we examined one way to sort lists. Here is a second way, which you should treat as independent of that previous question.

```
(define (sort lst)
  (if (null? lst)
      nil
      (let ((more (filter (lambda (x) (> x (car lst))) (cdr lst)))
            (less (filter (lambda (x) (<= x (car lst))) (cdr lst))))
        (append (sort less)
                (list (car lst))
                (sort more))))))
```

Suppose we want to generalize our methods to that they can work on list of arbitrary data structures, and with arbitrary comparisons.

Here is a version of such a procedure:

```
(define (general-sort lst lessthan extract)
  (if (null? lst)
      nil
      (let ((more (filter EXP1 (cdr lst)))
            (less (filter EXP2 (cdr lst))))
        (append (sort less)
                (list (car lst))
                (sort more))))))
```

The parameter `extract` is used to select the portion of each element of the list that is to be compared, and the parameter `lessthan` is to be applied to such extracted portions, returning a true value if and only if the first extracted portion is “less than” the second one.

**Question 25.** Write the code fragment for EXP1.

**Question 26.** Write the code fragment for EXP2.

Now suppose we want to sort a list of vectors, where a vector is defined by the following abstraction:

```
(define (make-vector x y)
  (list x y))
(define xcoord car)
(define ycoord cadr)
```

**Question 27.** Assume that `test-vectors` is a list of vectors. What application of `general-sort` will return a version of this list, sorted by increasing order of their `x` coordinates?

**Question 28.** Suppose our goal is to sort `test-vectors` so that they are arranged in increasing value of the `x` coordinate, and if the `x` coordinates are the same, then in increasing value of `y` coordinate.

What application of `general-sort` will do this?

**BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS**

**List Procedures**

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

```
(define (map op lst)
  (if (null? lst)
      nil
      (cons (op (car lst))
            (map op (cdr lst)))))
```

```
(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init (cdr lst)))))
```

**BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS**